Fermilab

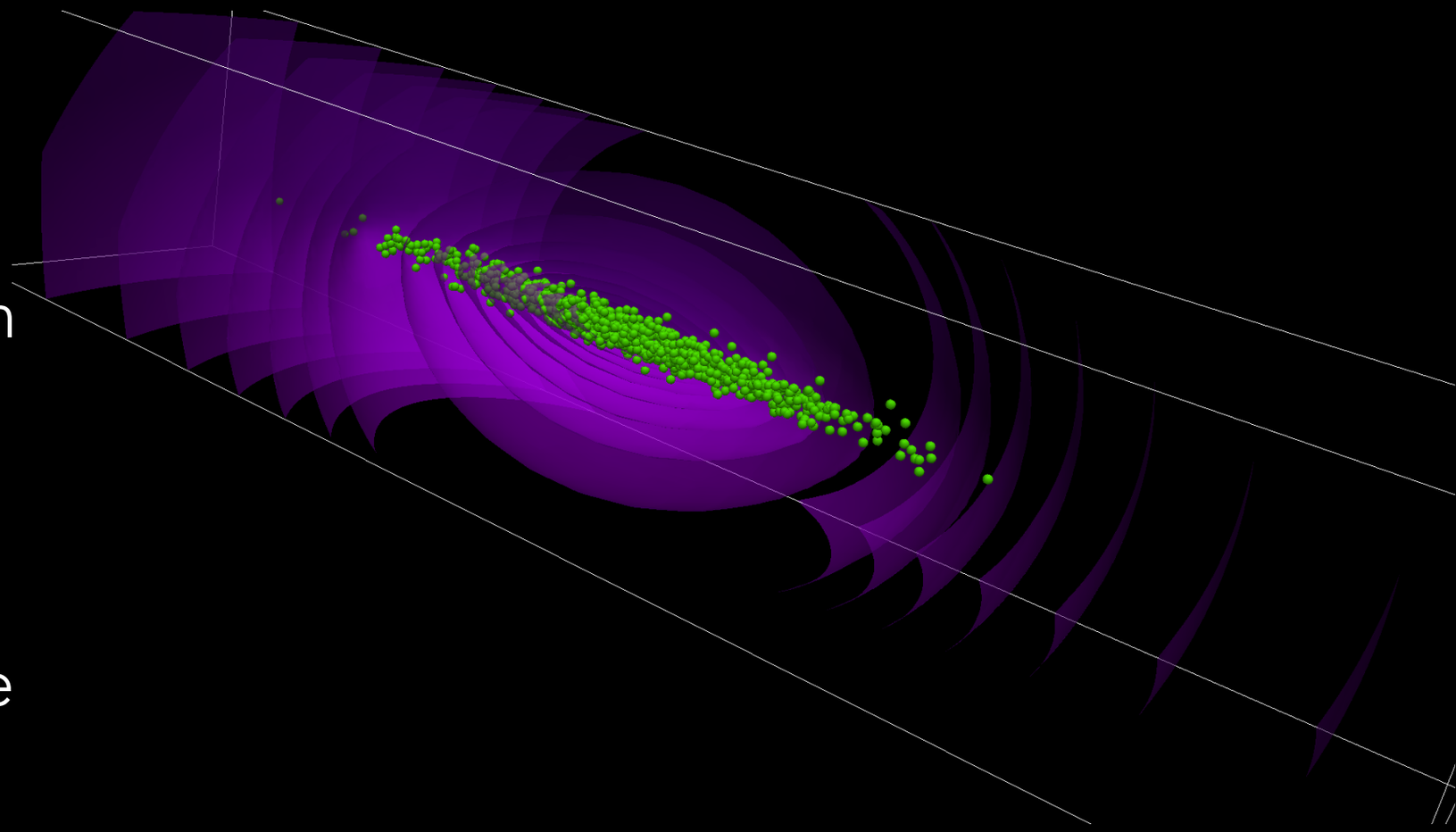
# **Synergia**: Driving Massively Parallel Particle Accelerator Simulations with Python

Qiming Lu, James Amundson and Eric Stern
Fermi National Accelerator Laboratory

WHAT: SYNERGIA

# ACCELERATOR SIMULATION PACKAGE

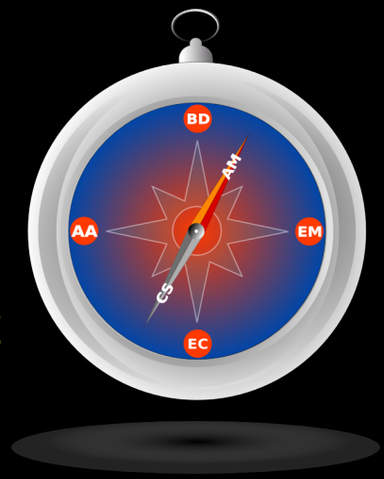
- Designed for range of computing resources
  - Laptops and desktops
  - Clusters
  - Supercomputers
  - GPU/Intel MIC acceleration platform
- Beam dynamics
  - Independent-particle
  - Collective effects



WHO: PERSONNEL

# SYNERGIA

**ComPASS**
A SciDAC project




is developed and maintained by the

## ACCELERATOR SIMULATION GROUP
## SCIENTIFIC COMPUTING DIVISION
## FERMILAB

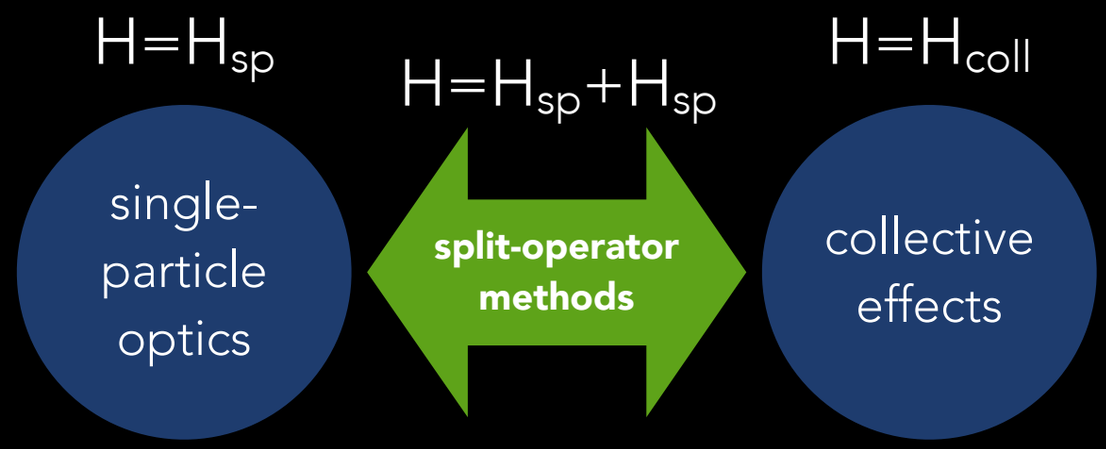
James Amundson, Paul Lebrun, Qiming Lu, Alex Marcidin,
Leo Michelotti (CHEF), Chong Shik Park, (Panagiotis Spentzouris),
and Eric Stern

# PHYSICS

- Single-particle physics
  - Provided by CHEF C++ library
  - Direct symplectic tracking, and/or arbitrary-order polynomial maps
- Apertures
- Collective effects (single and multiple bunches)
  - Space charge
  - Wake fields
  - Electron cloud, Beam-beam to be developed

WHAT: SYNERGIA

# TECHNIQUES




- Split-Operator

Allows to approximate the evolution operator for a time *t* by

$$\mathcal{O}(t) = \mathcal{O}_{sp}(t/2)\mathcal{O}_{coll}(t)\mathcal{O}_{sp}(t/2)$$

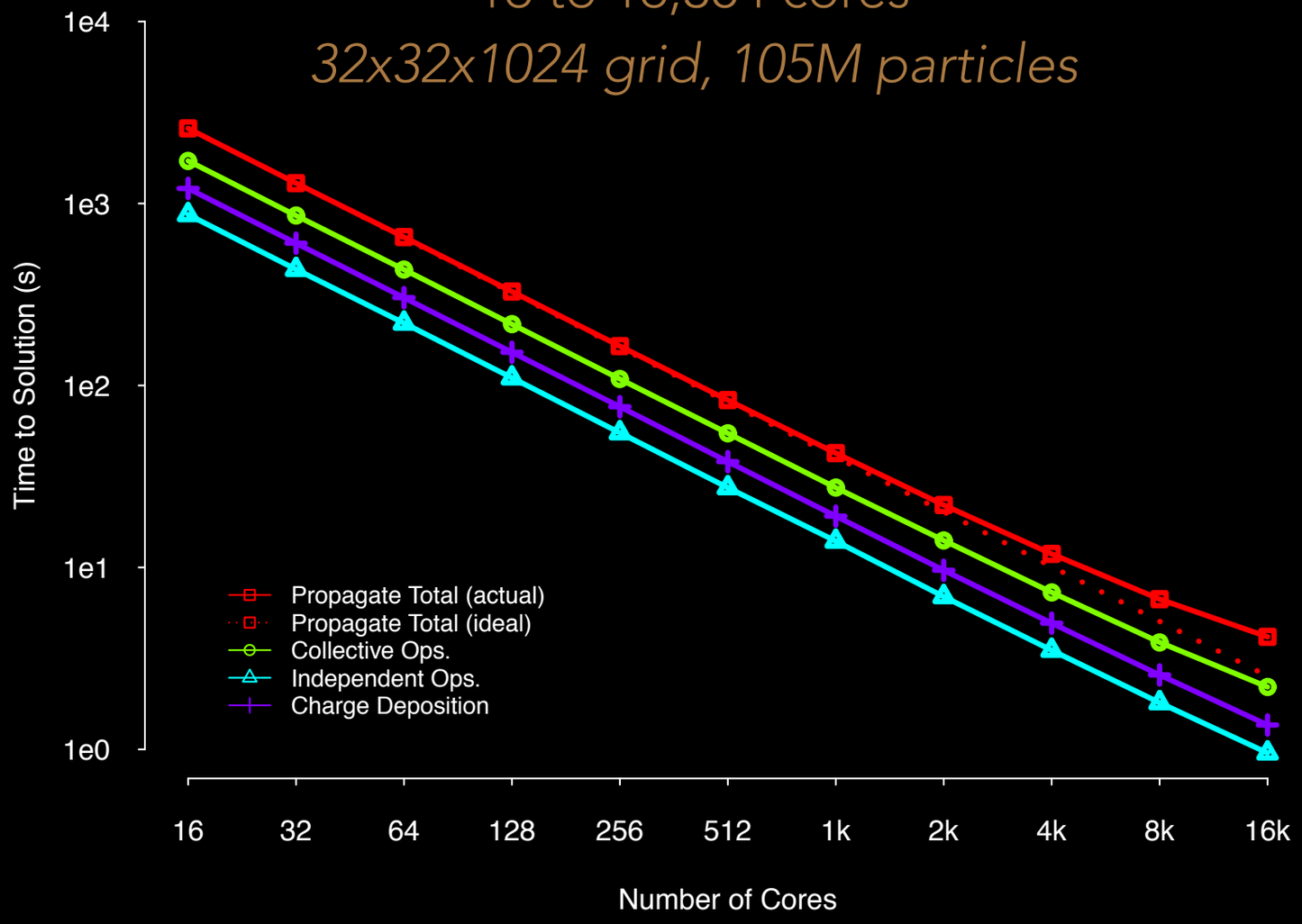
- Particle-in-Cell

Allows to simulate the large number of particles in a bunch (typically $O(10^{12})$) by a much smaller number of macro-particles ($O(10^{7})$). Collective effects are calculated using fields calculated on discrete meshes with $O(10^{6})$ degrees of freedom

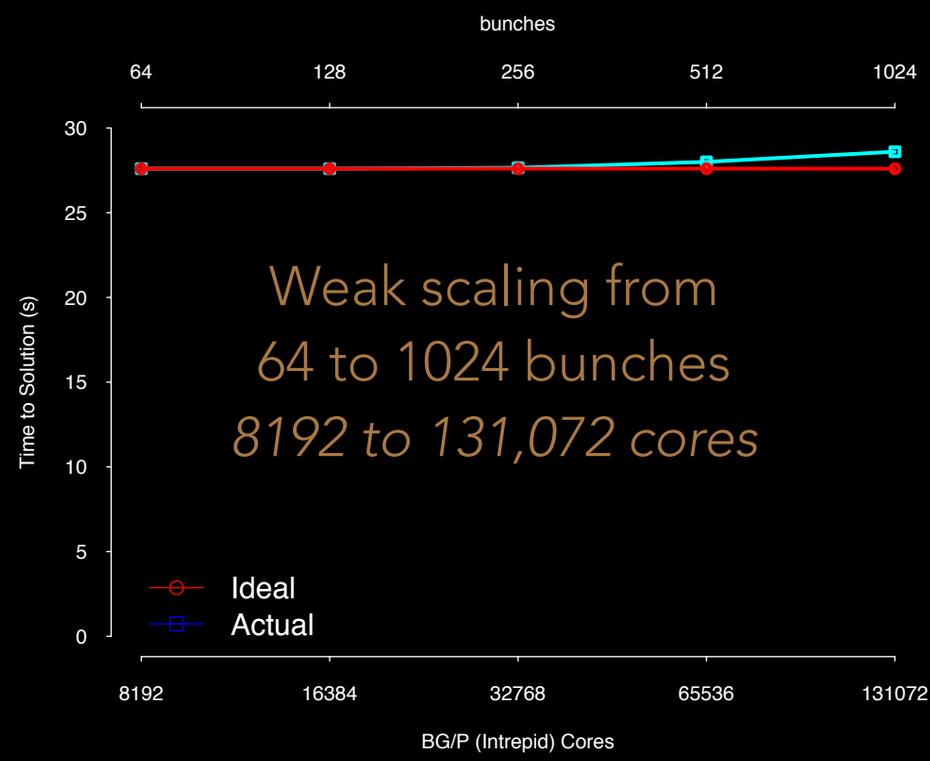
WHAT: SYNERGIA

# PARALLEL SCALING

Single-bunch strong scaling
16 to 16,384 cores
*32x32x1024 grid, 105M particles*

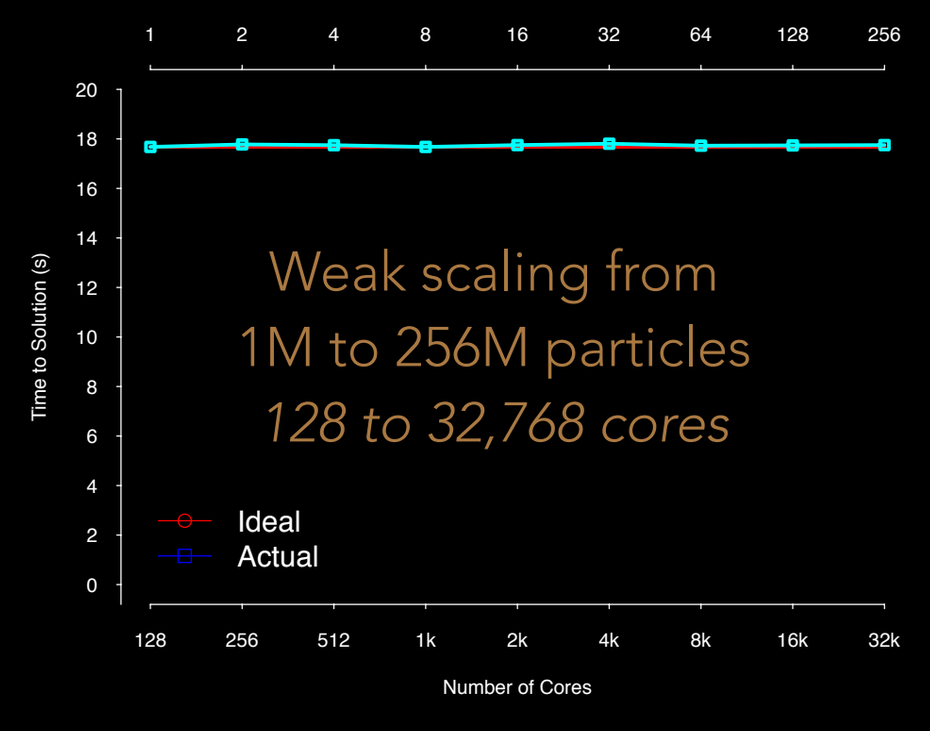



Benchmarks on ALCF's
Intrepid (BG/P) and Mira (BG/Q)



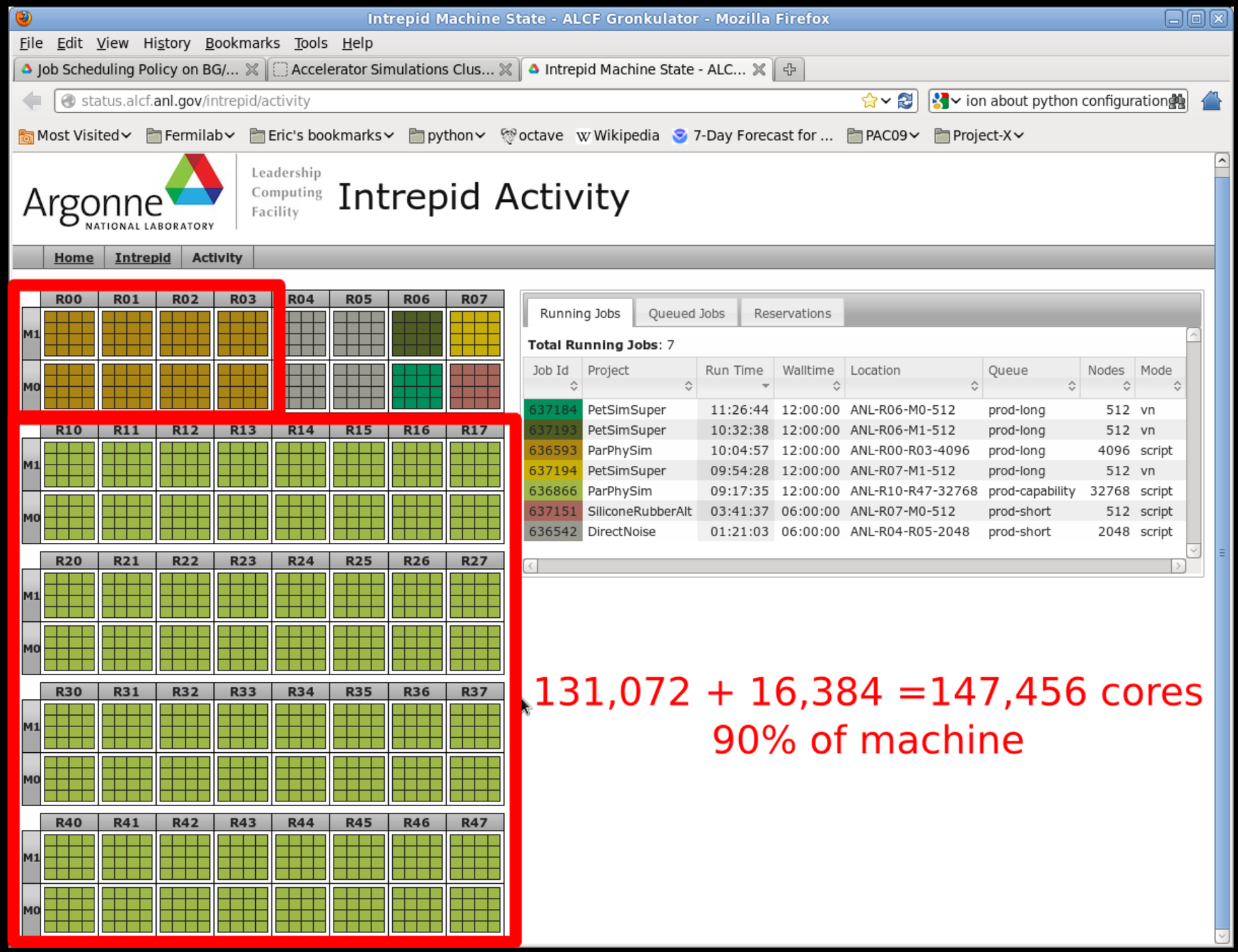

Weak scaling from
64 to 1024 bunches
*8192 to 131,072 cores*

Weak scaling from
1M to 256M particles
*128 to 32,768 cores*



## WHAT: SYNERGIA

# PRODUCTION



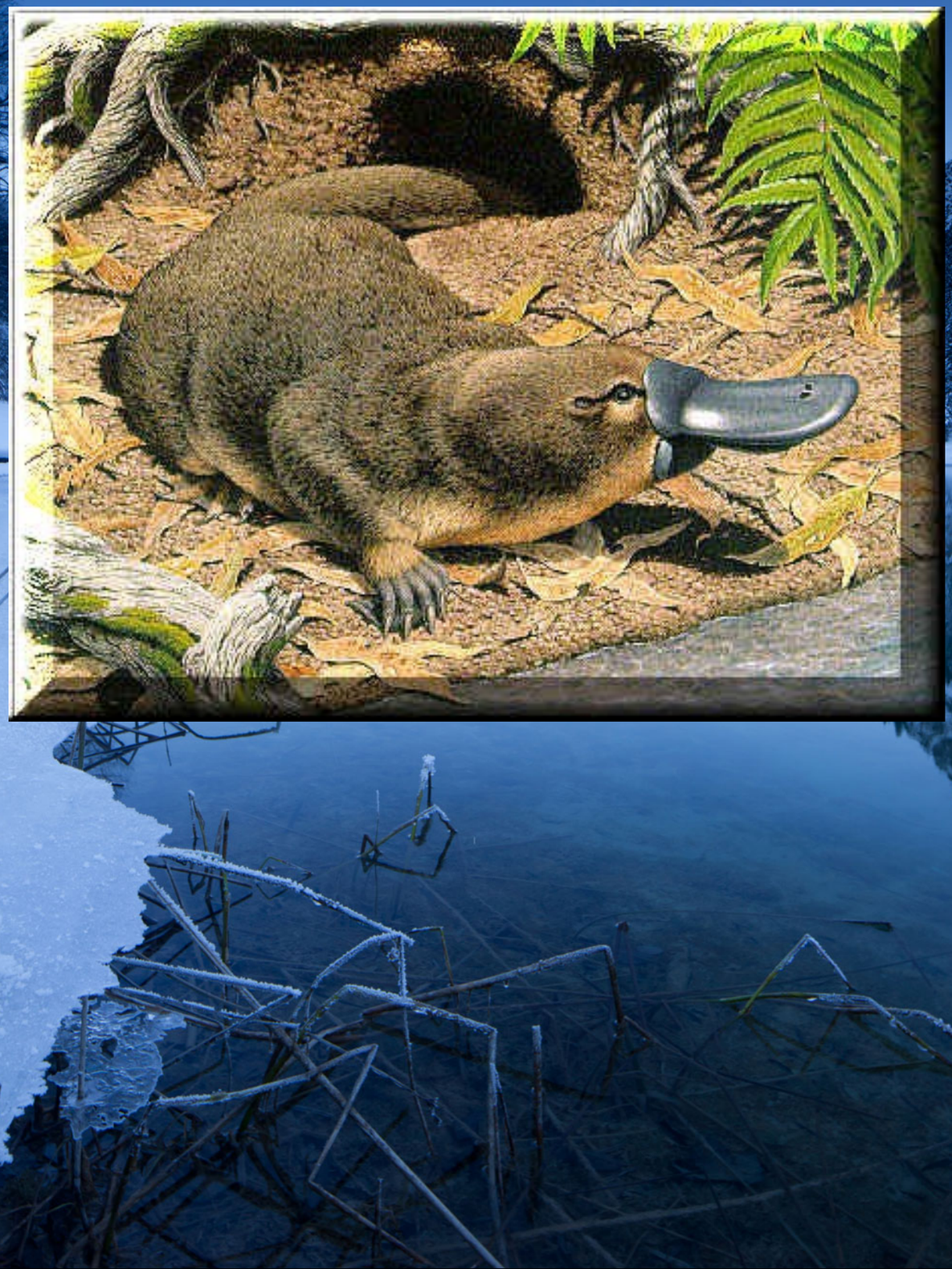
WHAT: SYNERGIA

# EVOLUTION

- Synergia 1 was a combination of IMPACT (F90) and CHEF (C++) + fronted (Python)

- Synergia 2 started as a proof-of-concept for a Python driver. Design heavily influenced by IMPACT and Fortran. Evolved its own C++ space charge solvers

- Synergia 2.1 developed into a robust, pure-C++ class structure with a Python-C++ interface



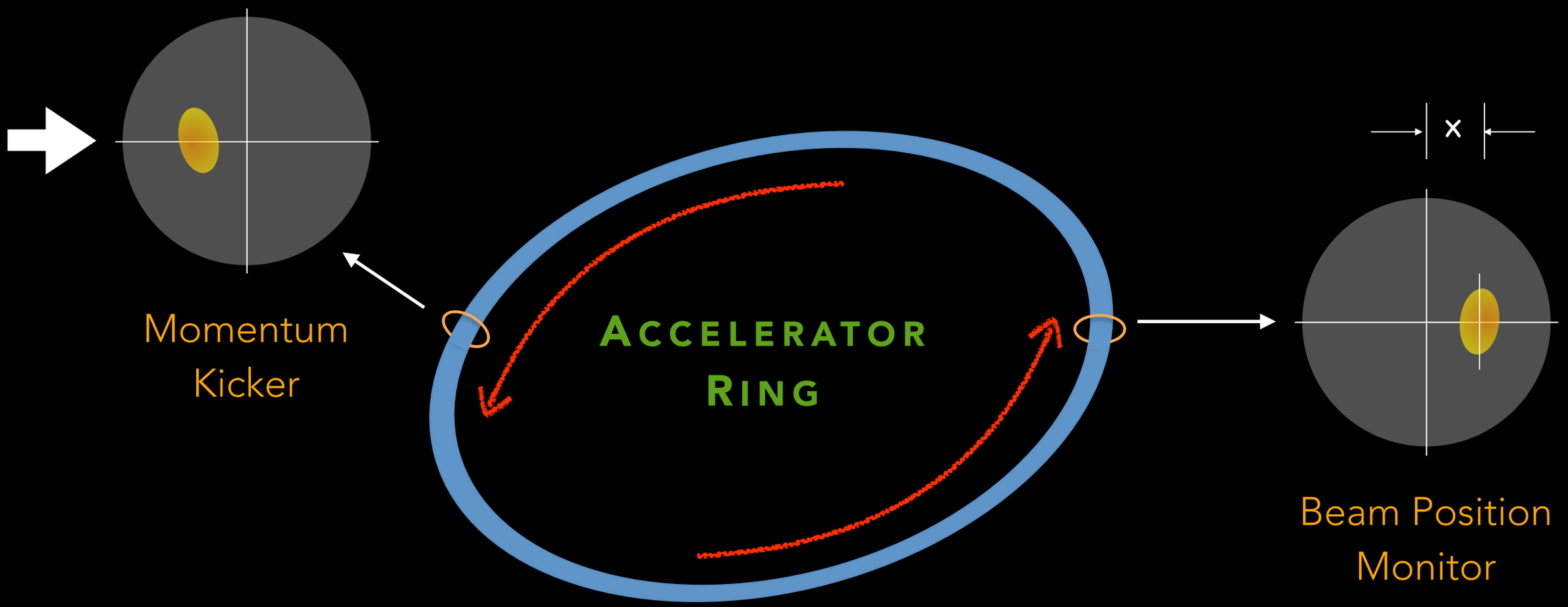
# WHY PYTHON?

- Realistic accelerator simulations require a very complicated description of simulation parameters
- Using Python as a driver gives the end user access to a full programming language to use in simulation description
  - We did not have to develop and maintain a new language
  - End users do not have to learn YASPL (yet another single-purpose language)
- Capable of creating dynamic simulations including time-varying machine parameters and active feedback

WHY PYTHON

# Example 1 — Active Damping

Toy example of real-world application



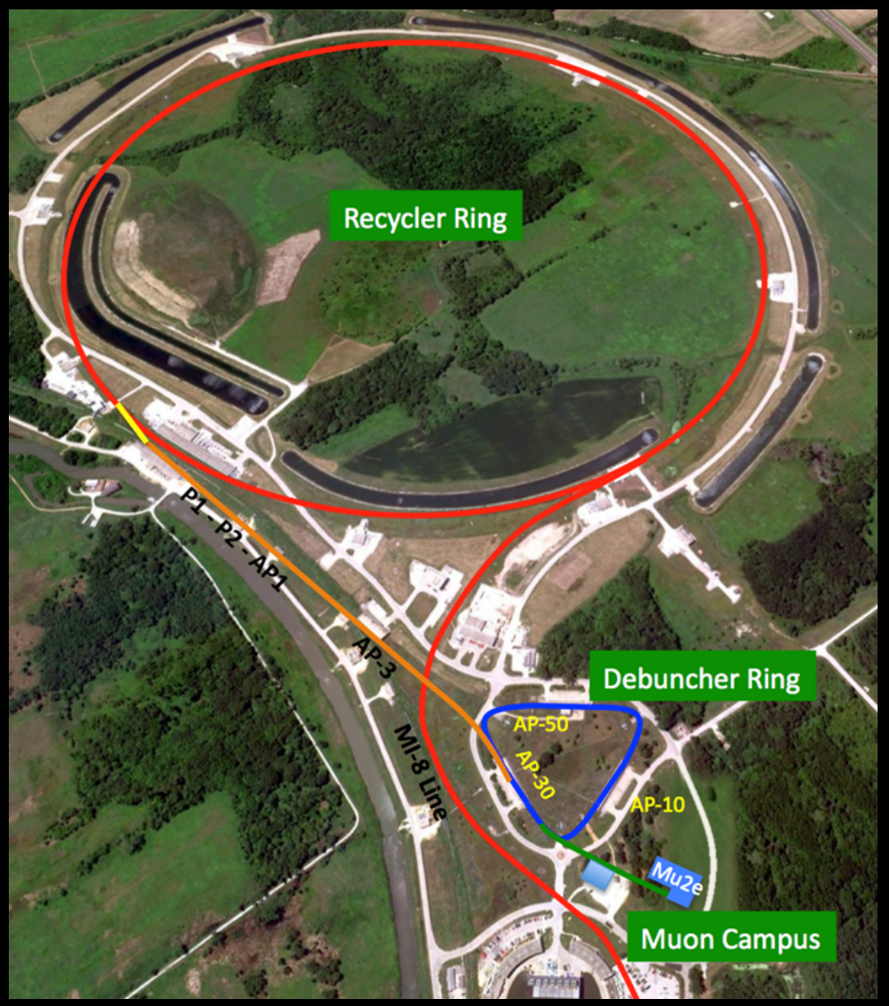

Counteract instabilities induced by wakefield

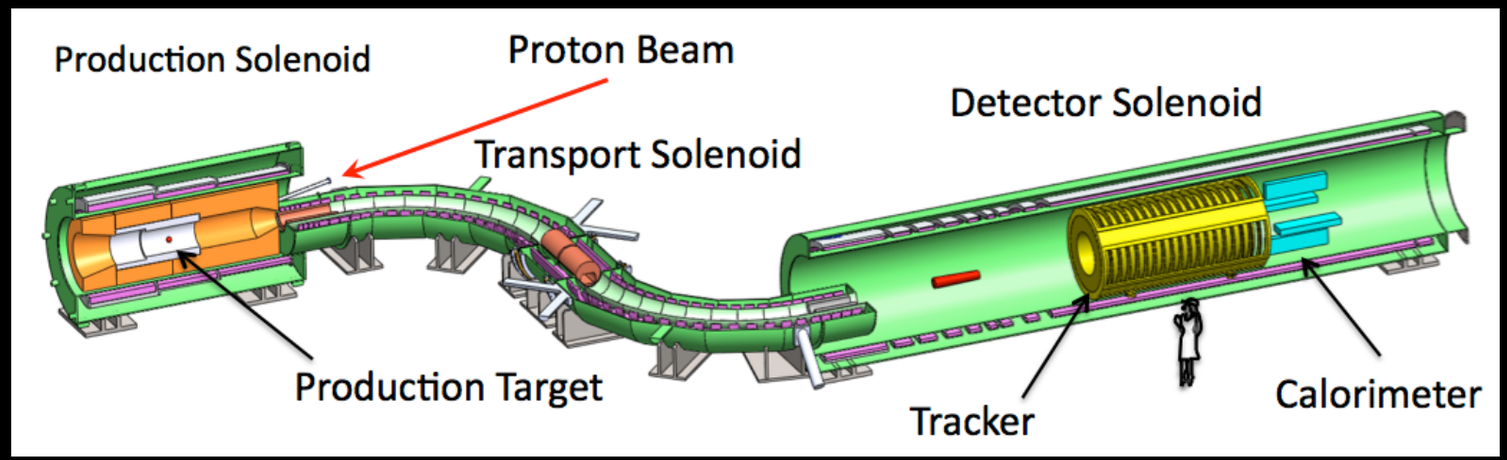
WHY PYTHON

# Example 1 — Active Damping

```
from synergia import Propagate_actions, Pickle_helper,
    bunch.Core_diagnostics

class Damper_actions( Propagate_actions, Pickle_helper ):
    def __init__(self, bpm_location, damper_location):
        Propagate_actions.__init__(self)
        Pickle_helper.__init__(self, bpm_location, damper_location)
        self.bpm_location = bpm_location
        self.damper_location = damper_location
        self.bunchx = 0.0
        # ...

    def step_end_action(self, stepper, step, bunch, turn_num, step_num):
        # Measure the bunch position at the pickup (BPM) location
        if step_num == self.bpm_location:
            self.bunchx = Core_diagnostics().calculate_mean(bunch)[0]

        # Shift the bunch position at the damper location
        elif step_num == self.damper_location:
            # kick x momentum to restore position
            bunch.get_local_particles()[:, 1] += -gain*bunchx/self.betax
```

# EXAMPLE 2 — MU2E




- Real example of real-world application
- Resonant extraction scheme for mu2e experiment at Fermilab
- Extraction for experiment requires ramping of magnet strength to move selected particles onto resonance

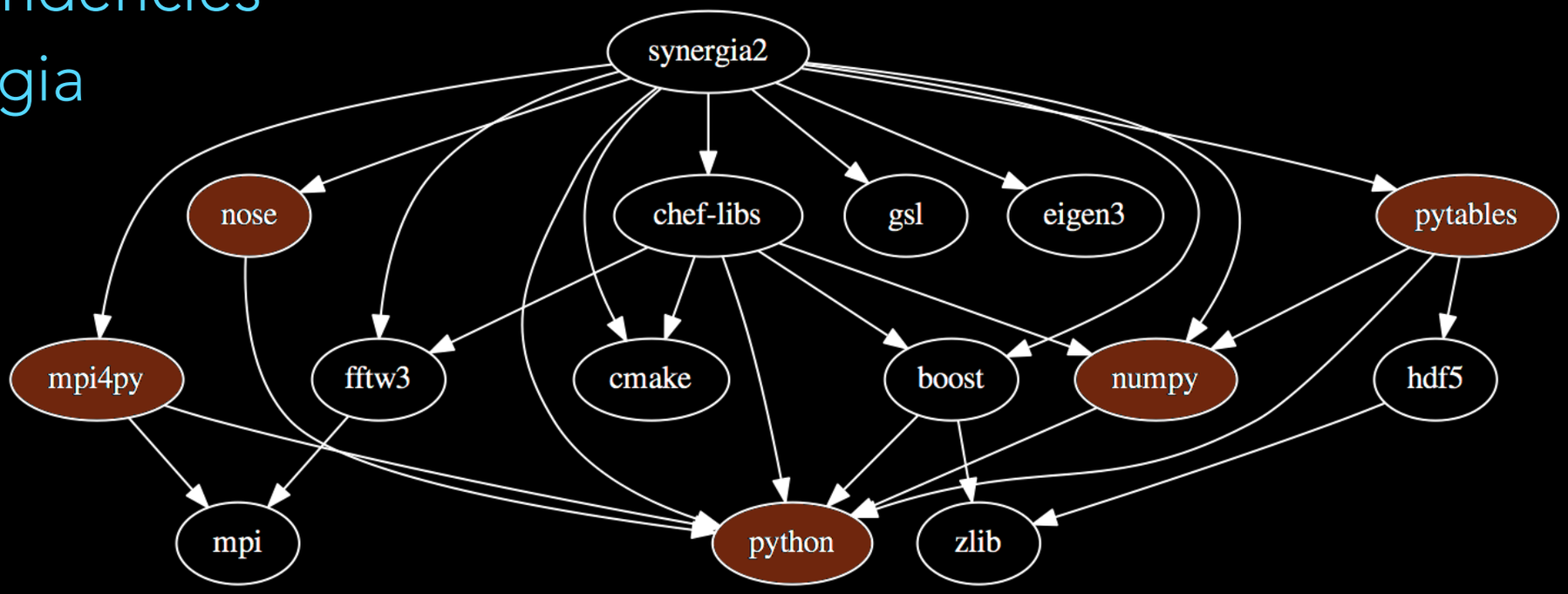
# WHY PYTHON EXAMPLE 2 — MU2E

```
from synergia import Propagate_actions, Pickle_helper

class Ramp_actions(Propagate_actions, Pickle_helper):
    def __init__(self, ramp_turns, turns_to_extract, inital_k1,
                 final_k1, final_k21, rfko_kicker):
        Propagate_actions.__init__(self)
        Pickle_helper.__init__(self, ramp_turns, turns_to_extract,
                               initial_k1, final_k1, final_k21, rfko_kicker)
        # ...

    def turn_end_action(self, stepper, bunch, turn_num):
        synergia_elements = \
            stepper.get_lattice_simulator().get_lattice().get_elements()
        # ...

        if turn_num <= self.ramp_turns:
            index = 0
            for element in synergia_elements:
                if element.get_type() == "multipole":
                    new_k21 = self.final_k21[index]*turn_num / \
                        self.ramp_turns
                    element.set_double_attribute("k21", new_k21)
                    index += 1
            if turn_num == 1:
                old_intensity = bunch.get_total_num()
                n0 = old_intensity
                avg_rate = n0 / float(self.turns_to_extract - self.ramp_turns)
        # ...
```

SYNERGIA

# WHY NOT PYTHON?

## Package dependencies of Synergia




## Portability

Shared libraries

Everyone *says* they support shared libraries.
In practice, shared versions of installed libraries rarely available on HPC platforms

Cross-compilation of Python

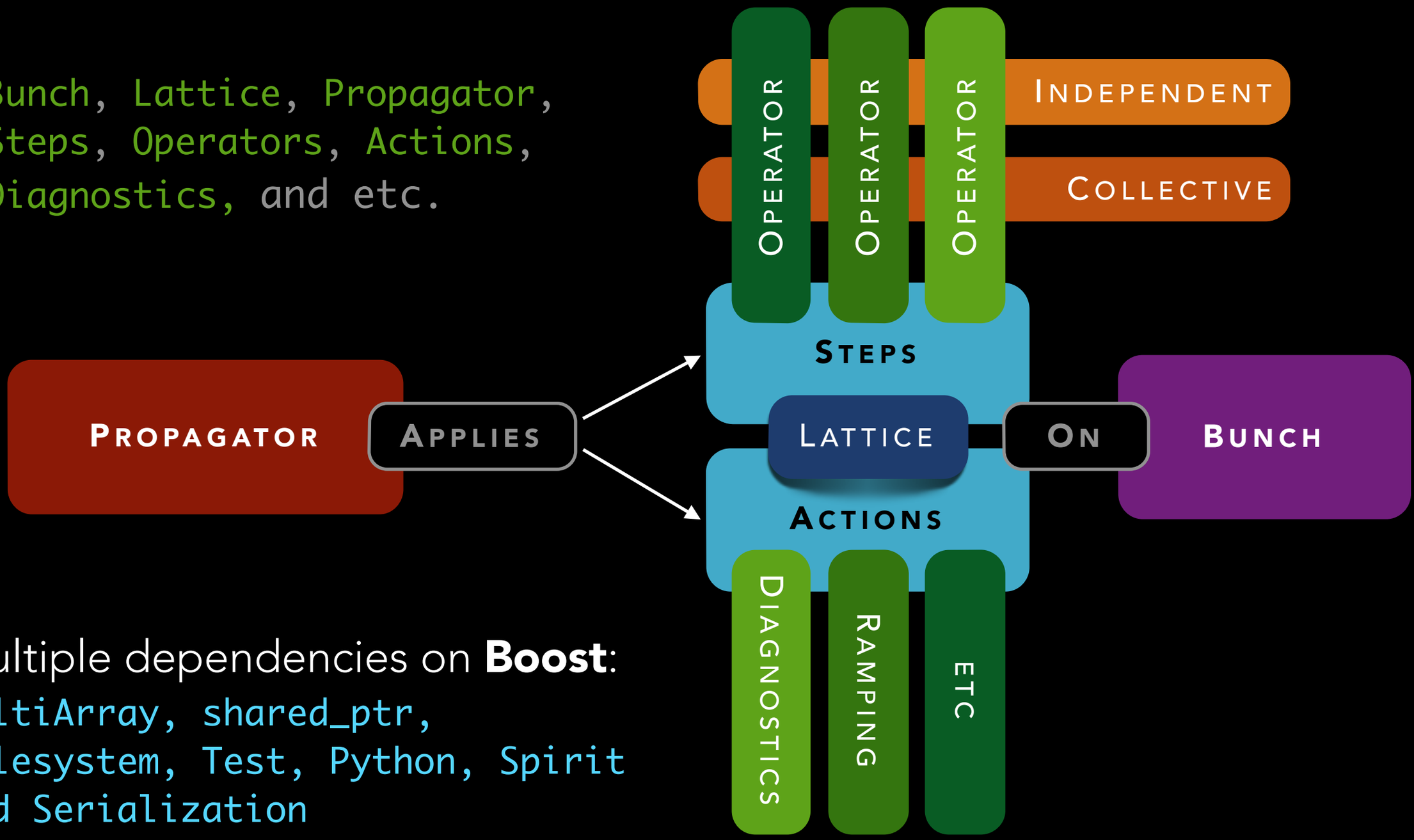
## Debugging / Profiling

There are tools which support the mixed Python-C++ application, but choices are very limited

SYNERGIA

# C++ Core Classes

Bunch, Lattice, Propagator,
Steps, Operators, Actions,
Diagnostics, and etc.




Multiple dependencies on **Boost**:
MultiArray, shared_ptr,
Filesystem, Test, Python, Spirit
and Serialization

SYNERGIA

# Python-C++ Interface

## Boost Python: expose C++ interfaces to Python

- Classes can be wrapped straightforwardly, preserving inheritance relations. Python classes can inhered from C++ classes
- C++ classes can call inherited methods in Python classes through callbacks
- Overloaded C++ signatures can be translated to Python with a minimum of effort
- Classes and shared pointers to classes can be interchanged transparently at the Python level

SYNERGIA

# Python-C++ Interface

- C++ code looks like C++ code
- Python code looks like Python code
- C++ containers can be converted to native Python containers with ease
  - `std::vector` and `std::list` ==> Python `list`
  - `Boost.MultiArray` ==> numpy arrays
- Minimal overhead, no need to copy the underlying data

# PYTHON-C++ INTERFACE

- MPI communicators are handled through a thin wrapper in C++ called `Commxx`

- In first version of Synergia mpi4py communicators are convertible to/from `Commxx` objects

- `Commxx` evolves to accommodate the need for serializing/ reconstructing MPI communicators

- in python scripts, mpi4py calls `MPI_Init()` / `MPI_Finalize()`

# SERIALIZATION

- **Checkpointing** is an important feature for large-scale simulation applications

- Checkpointing by re-initialization is impractical for Synergia
  - Synergia simulations can be dynamic
  - it is really a library that end-users use to create their own applications

- Synergia uses an object serialization scheme that allows the state of each of its object (including user defined Python objects derived from Synergia) to be written to and/or restored from disk

SYNERGIA

# SERIALIZATION

## C++ OBJECTS

BOOST.SERIALIZATION

## PYTHON OBJECTS

PYTHON.PICKLE

## USER DEFINED PYTHON OBJECTS DERIVED FROM PYTHON WRAPPED C++ OBJECTS

SYNERGIA.PICKLE_HELPER

1. PYTHON.PICKLE: PYTHON OBJECT -> PICKLE STRING
2. BOOST.SERIALIZATION: PICKLE STRING -> BINARY OR XML

(FULLY AUTOMATED)

SYNERGIA

# SERIALIZATION

```
template<class Archive>
    void save(Archive & ar, const unsigned int version) const
    {
        ar &
        BOOST_SERIALIZATION_BASE_OBJECT_NVP(Propagate_actions);
        std::string pickled_object(
            extract<std::string>(import("cPickle").attr("dump")(self)));
        ar & BOOST_SERIALIZATION_NVP(pickled_object);
    }

template<class Archive>
    void load(Archive & ar, const unsigned int version)
    {
        ar &
        BOOST_SERIALIZATION_BASE_OBJECT_NVP(Propagate_actions);
        std::string pickled_object;
        ar & BOOST_SERIALIZATION_NVP(pickled_object);
        str pickle_str(pickled_object);
        self = import("cPickle").attr("loads")(pickle_str);
    }

BOOST_SERIALIZATION_SPLIT_MEMBER()
```

BOOST.SERIALIZATION IN PROPAGATE_ACTIONS

SYNERGIA.PICKLE_HELPER

```
class Pickle_helper:
    __getstate_manages_dict__ = 1
    def __init__(self, *args):
        self.args = args
    def __getinitargs__(self):
        return self.args
    def __getstate__(self):
        return self.__dict__
    def __setstate__(self, state):
        self.__dict__ = state
```

“Python is wonderful.”